Concretização de uma Arquitectura de Suporte à Geração de Cenas Animadas com Agentes Inteligentes

Miguel Silvestre
Maria Pinto-Albuquerque
Maria Beatriz Carmo
Ana Paula Cláudio
Helder Coelho

DI-FCUL TR–05–6

March 2005

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749-016 Lisboa
Portugal

Technical reports are available at http://www.di.fc.ul.pt/tech-reports. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

Concretização de uma Arquitectura de Suporte à Geração de Cenas Animadas com Agentes Inteligentes

Miguel Silvestre[1]
Maria Pinto-Albuquerque[2]
Maria Beatriz Carmo[1]
Ana Paula Cláudio[1]
Helder Coelho[1]

[1]Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
1749-016 LISBOA, Portugal
i25293@alunos.di.fc.ul.pt,
{bc, apc, jdc, hcoelho}@di.fc.ul.pt

[2]Departamento de Ciências e Tecnologias da Informação
ISCTE
1649-026 LISBOA, Portugal
maria.albuquerque@iscte.pt

Resumo

Este relatório descreve a arquitectura e opções tomadas no projecto MAGO2 relativamente à tarefa sobre animação de agentes inteligentes. A arquitectura deverá permitir a criação de ambientes virtuais onde se movimentam agentes inteligentes. Esta arquitectura faz a articulação de várias aplicações – software de modelação tridimensional (Blender), motor gráfico (Ogre), motor físico (ODE) e bancada de agentes (JADE).

1 Introdução

O projecto MAGO2, Modelação de Agentes em Organizações, integra várias tarefas relacionadas com a estrutura e dinâmica de sociedades de agentes. Uma das tarefas incorpora uma vertente de representação gráfica. O objectivo desta tarefa é a concretização de uma biblioteca de representações gráficas para agentes inteligentes com a forma de humanóide com expressões faciais capazes de transmitir emoções e estados de espírito. A existência desta biblioteca tornará possível a realização de experiências que permitam simular o comportamento de agentes individualmente ou integrados em sociedade.

Este documento vem na sequência do relatório técnico anteriormente publicado [Silvestre04] e nele se focam a arquitectura adoptada e as ferramentas escolhidas para a sua concretização.

O documento encontra-se organizado do modo seguinte: na secção 2 descreve-se sucintamente a arquitectura proposta; a bancada de agentes escolhida é descrita na secção

3; a secção 4 é relativa à ligação entre o processamento gráfico e o processamento de Inteligência Artificial (IA); a secção 5 debruça-se sobre o protocolo de comunicação; a secção 6 apresenta o esquema UML com a descrição das classes mais importantes que foram desenvolvidas.

2 A arquitectura

Para a concretização da tarefa em curso, como já foi referido em [Silvestre04], propôs-se uma arquitectura que se encontra reproduzida na figura Fig. 2-1.

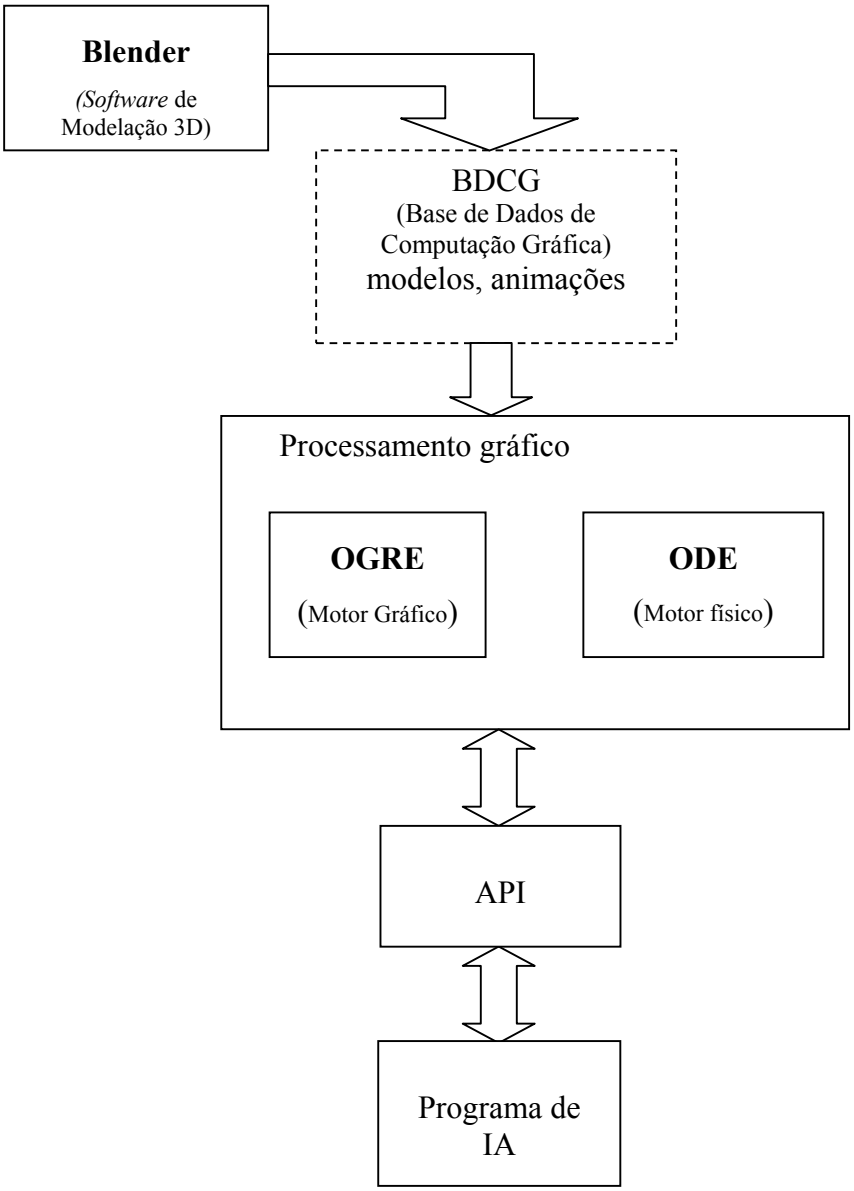


Fig. 2-1 Arquitectura usando *software* de modelação, motor gráfico e motor físico.

Com o software de modelação 3D criam-se os modelos gráficos dos agentes e as suas animações básicas, como andar, correr, entre outras. De seguida, estes modelos são exportados para um formato conhecido do motor gráfico (`*.mesh`), criando-se assim uma Base de Dados de modelos animados ou inanimados (como é o caso dos elementos estáticos, cadeiras, mesas, objectos de arte). O motor gráfico é responsável por todo o *rendering* da cena 3D, e o motor físico é responsável pela simulação física dos objectos. O motor físico para além de simular as colisões entre objectos, também permite a definição de massas, acelerações, gravidade, entre outros. Para já, apenas lidamos com a simulação da gravidade e de colisões.

Estas componentes (motor gráfico e motor físico) comunicam com o programa que faz o processamento de IA ou um *Multi Agent System (MAS)* através de uma API (*Application Program Interface*).

O *software* escolhido para a arquitectura foi:

- Modelação gráfica – Blender [blender]
- Motor gráfico – Ogre [ogre]
- Motor físico – ODE [ode]

Uma descrição deste *software* pode ser encontrada em anexo.

2.1 O cenário

Os elementos que irão constituir o cenário foram agrupados em três grupos distintos: agentes, itens e paredes. Foram considerados estes três grupos pois a forma como serão tratados pelo sistema inteligente será distinta. A definição destes grupos torna a procura de elementos do cenário mais eficiente. Por exemplo, quando é necessário saber quais os agentes presentes no cenário basta pedir a lista de agentes.

Os agentes que estarão presentes no cenário tridimensional exibem comportamentos inteligentes e tipicamente têm uma forma humanóide. Um agente tem vários sensores associados: visão, audição, olfacto. Neste momento apenas a visão se encontra implementada. O agente poderá também ter diversas animações associadas: andar, correr, virar a cabeça, sorrir, entristecer, entre outras.

As paredes estão separadas do resto do cenário pois não existe necessidade de guardar muita informação sobre as mesmas para além da necessária para os agentes evitarem colidir com elas.

Tudo o resto é considerado um item. Se pensarmos na visita a um museu como exemplo de aplicação, os itens podem ser placas indicativas (placas de saída, placas a indicarem para que lado se encontram certos objectos de arte), ou objectos de arte como quadros, ou sofás, cadeiras, ou mesas, entre outros. Estes itens têm propriedades próprias que é necessário guardar. Quando se trata de um item de interesse para o agente é necessário saber que tipo de item é (por exemplo, um quadro, uma peça de arte) e quais as suas características (por exemplo: época, estilo). Alternativamente, o item pode não ter interesse nenhum para o agente e neste caso a única informação a registar é a sua posição. Imagine-se um cenário em que o agente anda num museu mas não se pode sentar. Neste caso as cadeiras existentes no museu têm muito pouco interesse para o agente pois este não as pode utilizar. A única informação com interesse para o agente é a posição da cadeira para não colidir com ela.

2.2 Ligação entre as componentes do processamento gráfico

Para garantir a ligação entre as aplicações Ogre e ODE desenvolvemos, sobre estas, uma camada de software designada por camada do Processamento Gráfico. Esta é composta por um conjunto de classes designadas por classes do Processamento Gráfico (PG).

Descrevemos nesta secção como estas classes, as do Ogre e as do ODE são usadas para se obter a representação gráfica de uma cena.

Todos os elementos do cenário são modelados no Blender e de seguida exportados para o formato *.mesh* para que o Ogre possa produzir o *rendering* correspondente. Cada ficheiro **.mesh* é "carregado" para o motor gráfico recorrendo à classe *Entity* do Ogre (*Ogre::Entity*). De seguida, para que a imagem correspondente ao conteúdo deste ficheiro seja apresentada no ecrã e adicionada à cena, é necessário associá-la a um objecto da *SceneNode* do Ogre (*Ogre::SceneNode*). No caso de ser necessária a criação de propriedades de colisão ou físicas, também é necessário transformar as entidades existentes no Ogre em entidades que possam ser processadas pelo ODE.

O ODE permite a existência de várias geometrias de colisão: caixas ou paralelepípedos, cilindros, esferas, malhas poligonais, planos e raios (*rays*). Note-se que os planos e as malhas poligonais possuem apenas propriedades de colisão, não têm propriedades físicas. Quando se pretende associar propriedades de colisão a uma entidade é necessário criar uma das geometrias de colisão referidas e associá-la à correspondente entidade (objecto da classe *Entity*) do Ogre. As classes da camada do PG permitem a criação de todas estas formas geométricas, no entanto a mais usada é a caixa. O Ogre permite-nos muito rapidamente obter informação sobre a geometria de colisão de determinada entidade; com base nesta informação cria-se uma caixa invisível no ODE com propriedades de colisão e físicas correspondentes à entidade. Podemos imaginar um escudo impenetrável e invisível à volta da entidade, com a forma de uma caixa. Para tal existe na camada do PG a classe *PhysicObject* que faz precisamente a ligação entre o Ogre e o ODE, encapsulando os detalhes de criação do objecto ODE (para mais detalhe consultar a secção 6 e o anexo A).

Uma mesma entidade pode ter várias descrições diferentes consoante o tipo de tratamento que se pretende. Por exemplo, se pretendermos que uma *mesh* tenha propriedades físicas e de colisão, então é necessário criar um objecto do tipo *PG::PhysicObject*, criando-se assim uma descrição física para a entidade. Mais, se a *mesh* em causa representar um agente, então será criado um objecto do tipo *PG::Agent*, havendo portanto outra descrição da entidade. Portanto, se pretendermos um agente com propriedades físicas e de colisão temos que criar um objecto do tipo *SceneNode* no Ogre, um objecto do tipo *PG::Agent* e ainda um do tipo *PG::PhysicObject*. São três descrições distintas para a mesma entidade: a descrição gráfica (*Ogre::SceneNode*), a descrição do agente (*PG::Agent*) e a descrição física (*PG::PhysicObject*).

A classe *Entity* do Ogre tem um atributo do tipo *String*, *name*, que desempenha um papel fundamental na classificação dos vários elementos do cenário. Para as entidades que correspondem a paredes este atributo começa por "wall_"; no caso de agentes, este atributo começa por "agent_". Todas as outras entidades que não têm este atributo num destes formatos são considerados itens (por exemplo cadeiras, portas, mesas).

3 Bancada de agentes

A fim de simular os comportamentos inteligentes dos agentes optou-se por utilizar uma bancada de agentes já existente e bastante divulgada: JADE (*Java Agent DEvelopment Framework*).

O JADE é uma plataforma de agentes que facilita a tarefa de criação de novos agentes, bem como toda a comunicação feita entre agentes. É uma plataforma distribuída, sendo transparente qual o local onde o agente se encontra a executar. Deste modo podemos ter vários agentes a serem executados em máquinas distintas.

Desenvolveu-se uma camada de *software* sobre o JADE, designada por camada do Processamento de IA (PIA), de forma a estabelecer a ligação entre o JADE e a camada do PG . A próxima secção descreve o modo como foi feita esta ligação. As classes destas camadas encontram-se descritas na secção 6.

4 Ligação entre o processamento gráfico e o processamento de IA

Com a introdução do JADE, surgiram novos desafios no desenvolvimento do projecto. O JADE está implementado em JAVA, enquanto o Ogre e o ODE usam o C++. A solução adoptada para fazer a ligação entre a camada do PG e a camada do PIA passou pelo recurso a *sockets*.

As *sockets* trazem-nos algumas vantagens. Além de podermos ligar a camada do PG e a camada do PIA, também permitem ter o *rendering* num computador e a bancada de agentes noutro computador. Estes poderão estar em locais completamente distintos pois a conexão é feita através de *sockets* e, portanto, compatível com uma utilização em redes de computadores.

Como se pretende uma conexão simples entre a camada do PG e a camada do PIA, usa-se apenas uma ligação TCP. A ideia principal é ter um agente na bancada JADE responsável pela comunicação e coordenação entre os agentes JADE e a camada de PG. Outra hipótese é cada agente JADE ligar-se directamente à camada de PG, havendo assim tantas ligações quantos os agentes existentes. De momento, usa-se apenas uma ligação.

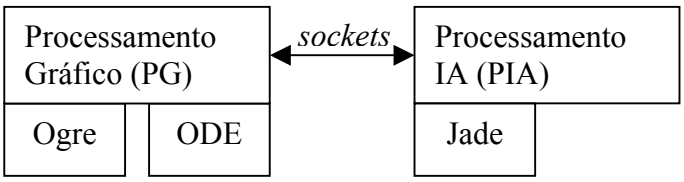


Fig. 4-1 Arquitectura usando o Blender como *software* de modelação, o OGRE como motor gráfico, o ODE como motor físico e o JADE como bancada de agentes.

5 Protocolo de comunicação

A introdução das *sockets* para fazer a ligação entre as camadas de PG e de PIA tornou necessário criar um protocolo de comunicação que fosse entendido por ambas as camadas. O protocolo ainda se encontra em fase de desenvolvimento. As duas camadas comunicam através de uma sequência de pedidos e respostas. A camada de PG efectua um pedido à camada de PIA, esta recebe o pedido, executa-o e envia a resposta à camada de PG. Desta forma temos um servidor TCP em C++ na camada de PG e um cliente TCP em Java na camada de PIA.

Considerou-se a hipótese de usar ficheiros XML para a troca de informação entre as bancadas. No entanto, com o intuito de optimizar o processamento e tratamento das mensagens que passam pelas *sockets*, optou-se pela utilização de *strings* de caracteres.

O protocolo é descrito nas duas subsecções seguintes.

5.1 Cliente

Sempre que um pedido é feito ao servidor, a mensagem começa com um dígito que depois será interpretado da seguinte maneira:

0 → o agente pergunta qual é o seu tamanho;

1 → o agente pergunta o que está a ver;

2 → o agente pergunta qual a sua posição;

3 → o agente pede para se deslocar. Neste caso é necessária mais informação como por exemplo, qual a direcção a tomar, velocidade e animação associada. Assim sendo estas mensagens terão o seguinte formato:

"3:animacao#velocidade#direccaoX#direccaoY#direccaoZ"=>"3:walk#80#0.5#0#0"

O carácter '#' serve como separador.

A fim de evitar muitas trocas de mensagens, sempre que um agente pede para se mover, o servidor, após efectuar o pedido, envia uma mensagem a dizer o que o agente está a ver. Não é pois necessário que o cliente efectue explicitamente o pedido 1.

5.2 Servidor

Quando o servidor recebe um pedido, interpreta a mensagem e executa as acções correspondentes devolvendo a resposta.

Os formatos das respostas às perguntas 0 e 2, qual o tamanho do agente e qual a sua posição, respectivamente, não diferem. A resposta é sempre composta por três valores reais, que num caso correspondem ao tamanho (altura; largura; e profundidade, por esta ordem) e no outro à sua posição (coordenada x, coordenada y, coordenada z, por esta ordem). As coordenadas estão separadas pelo carácter '#'.

Exemplo: "coordenada/tamanho X#coordenada/tamanho Y#coordenada/tamanho Z"
"34.9374275208#37.1535339355#8.08562088013"

A resposta à pergunta 1 é mais elaborada. Neste caso o agente pode estar a ver três tipos de objectos: outros agentes, paredes e itens. A fim de distinguir qual o tipo de

objecto que o agente está a ver, a mensagem tem um identificador seguido de ':'. O identificador pode ser 1, 2 ou 3 caso se trate de um agente, uma parede ou um item, respectivamente. A seguir ao carácter ':' vem a lista de objectos correspondentes separados pelo carácter '&'. O separador das listas é o carácter '%'. Mais uma vez, cada atributo do objecto em causa está separado por um '#'. A descrição do objecto em causa lista os seguintes atributos: primeiro vem o nome (identificador) do objecto, a distância a que este se encontra do agente (que perguntou o que está a ver) e de seguida as coordenadas onde se encontra o objecto.

Por exemplo a seguinte mensagem:

"2:wall_frontleft#63.6131401#-66.0938262#-3.7084865#6.62449896e-002&
wall_upper#61.3846588135#1.0855255127#22.2666435242#-61.2635726929
&%3:door#11.2072353363#0.107774019241#-12.5811138153#-0.412902951241&"

indica-nos que o agente vê paredes e itens (uma vez que não temos "1:" o agente não vê outros agentes). No caso das paredes ele vê duas: a "wall_frontleft" e a "wall_upper"; é também fornecida a distância das paredes ao agente, e a sua posição. O agente vê apenas um único item: "door".

6 Esquema UML e descrição das classes mais importantes

Nesta secção apresenta-se o esquema UML simplificado, e a descrição das classes das camadas do PG e do PIA.

Em anexo junta-se o esquema UML completo e a descrição detalhada das classes mais relevantes.

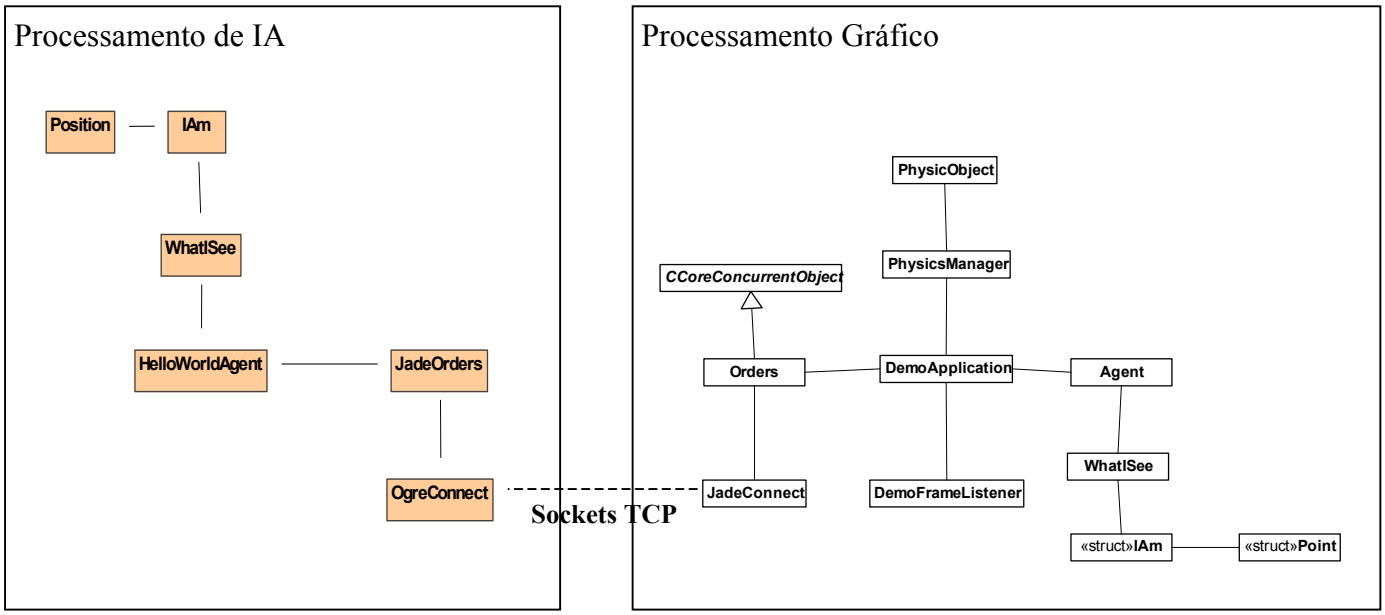


Fig. 6-1 Esquema UML simplificado

6.1 Descrição das classes do Processamento Gráfico

PhysicObject

Esta classe faz a ponte entre a *mesh* (malha poligonal) no Ogre e a sua descrição física no ODE. Sempre que se pretende associar uma simulação física ou de colisões a uma *mesh* é necessário criar um objecto desta classe.

PhysicsManager

PhysicsManager é a classe responsável pela criação e manutenção das instâncias da classe *PhysicObject*. Deverá existir apenas uma instância desta classe. Todas as instâncias de *PhysicObjects* que se desejam criar devem ser criadas através desta classe. Se eventualmente, se criar um objecto da classe *PhysicObject* sem usar a instância de *PhysicsManager* há que adicionar este objecto à lista desta classe que mantém um historial de todos os *PhysicObject* que estão criados.

JadeConnect

Classe responsável pela ligação, através de *sockets*, à camada do PIA. Como já foi dito considera-se a hipótese de existir apenas uma conexão do JADE ao Ogre em vez de várias. Esta classe desempenha o papel de servidor *socket* do sistema.

Esta classe é equivalente à classe *OgreConnect* do P IA.

Orders

Classe responsável pela criação e destruição do *JadeConnect*. Esta classe estende a *CCoreConcurrentObject* que cria uma nova *thread* que irá correr em ciclo no método *run*(). A classe *CCoreConcurrentObject* pretende encapsular a criação de novas *threads*, sendo a mesma transparente para o utilizador que crie a classe *Orders*.

Orders espera por mensagens vindas do cliente e processa-as.

Point

Apesar de em C++ esta classe ser uma estrutura é em tudo semelhante à classe *Position* da camada do PIA. Esta classe guarda três valores reais, um para cada coordenada x, y e z. Pode assim representar um ponto no espaço bem como a dimensão de uma caixa.

IAm

Estrutura simples que guarda certos dados importantes de um determinado elemento do cenário. É equivalente à classe *IAm* da camada do P IA.

WhatISee

Classe que guarda a informação sobre o que determinado agente vê. É equivalente à classe *WhatISee* pertencente à camada do PG.

Agent

Classe que representa um agente inteligente. Um agente tem certas características como os seus sensores (olhos), tamanho, etc. que estão representadas nesta classe.

DemoApplication

Esta é a classe inicial, onde tudo começa. Os ficheiros extraídos do Blender são carregados para o motor gráfico através de objectos desta classe. As câmaras, as opções de *rendering*, as luzes, etc. são todos inicializados através de objectos desta classe.

DemoFrameListener

Esta classe estende a classe *FrameListener* do Ogre. São invocados automaticamente métodos no início do *rendering* de uma imagem (método *frameStarted()*) e no fim (método *frameEnded()*). Usado para controlar os controlos dados pelos utilizadores e para actualizar as estatísticas que aparecem no écrã.

6.2 Descrição das classes de Processamento de IA

OgreConnect

Esta classe é semelhante à classe *JadeConnect* no sentido em que é responsável pelo estabelecimento da ligação da socket ao servidor. No entanto esta é uma socket de cliente e não de servidor.

JadeOrders

Classe responsável por enviar pedidos ao servidor e receber as respostas deste.

HelloWorldAgent

Os agentes que se constroem usando a bancada JADE têm que estender a classe Agent pertencente a essa bancada. Desta forma os agentes herdam as primitivas e funcionalidades básicas dos agentes da bancada JADE.

Como já foi referido *Position*, *IAM* e *WhatISee* são equivalentes, respectivamente às classes *Point*, *IAM,* e *WhatISee* do Processamento Gráfico.

7 Referências

[blender] http://www.blender3d.org/About/?sub=Features
"The Official Blender 2.3 Guide, The Open 3D Creation Suite",
Blender Foundation (eds) Amsterdam

[jade] http://jade.tilab.com/

[ode] http://ODE.org/

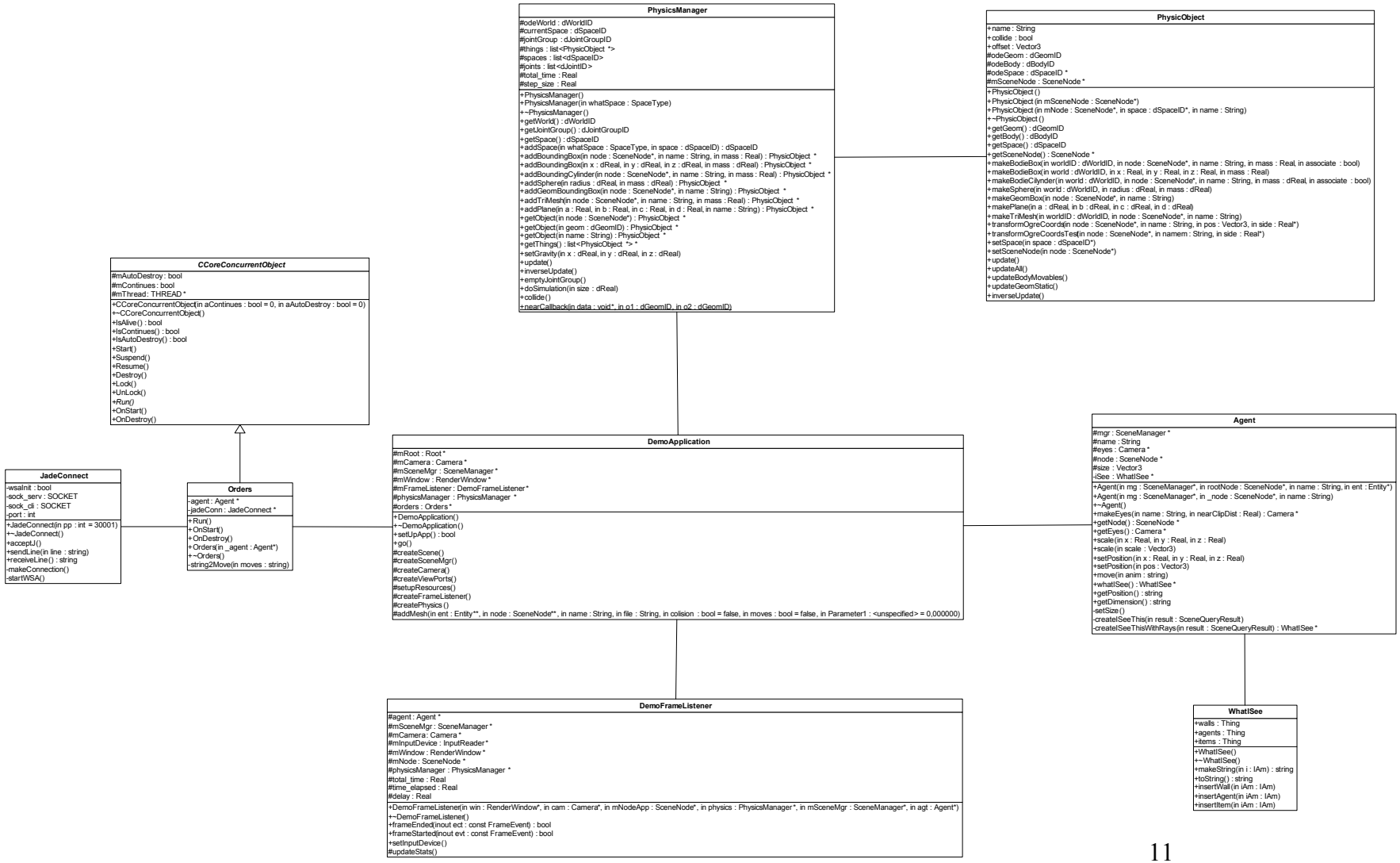
[Silvestre04] http://www.di.fc.ul.pt/tech-reports/abstract.php?report_ref=2004-07
"Arquitectura de Suporte à Geração de Cenas Animadas com Agentes Inteligentes", M. Silvestre, M. Pinto-Albuquerque, M. B. Carmo, A. P. Cláudio, J. D. Cunha, H. Coelho, Relatório Técnico do DI, TR-04-7, Julho 2004

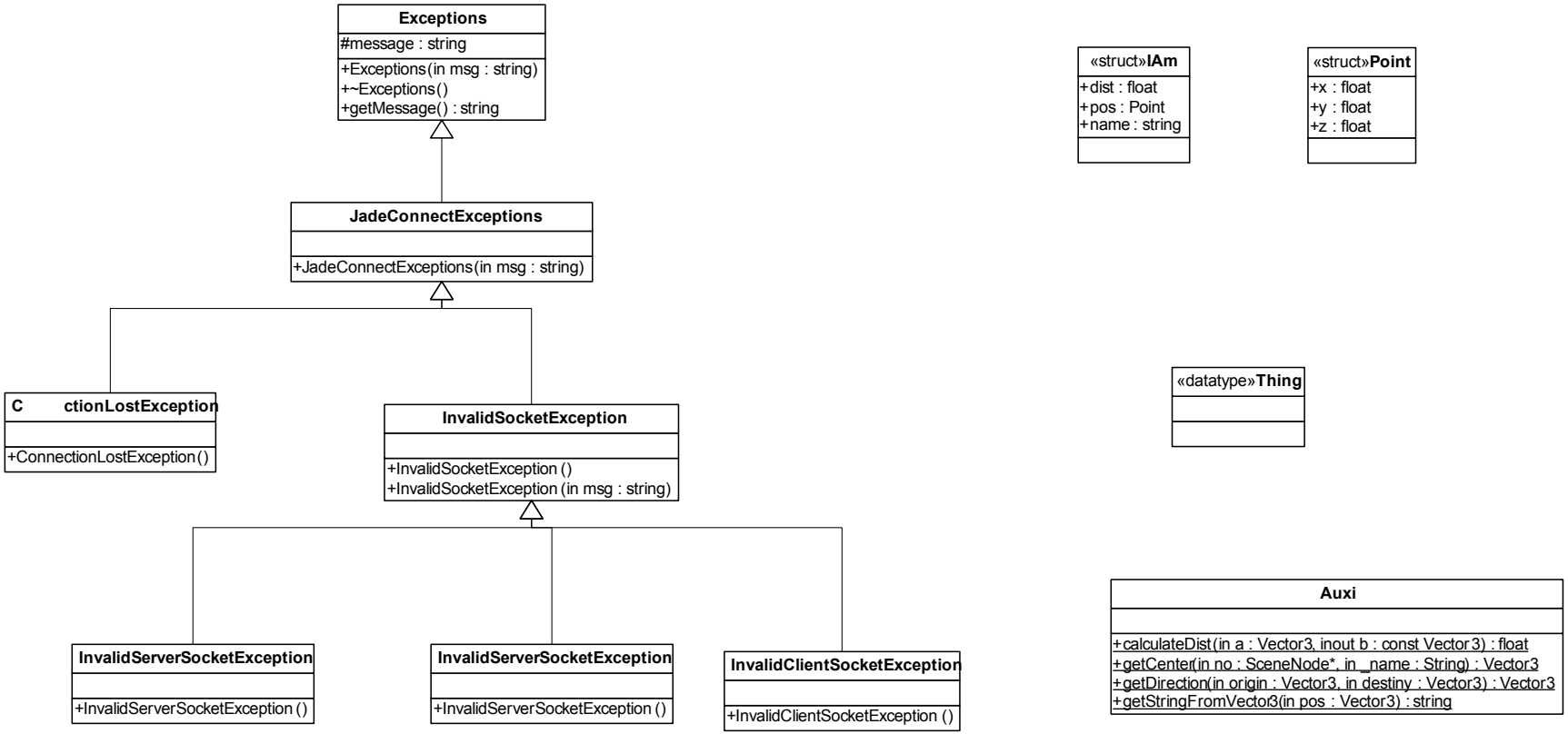
[ogre] http://www.ogre3d.org

Anexo A

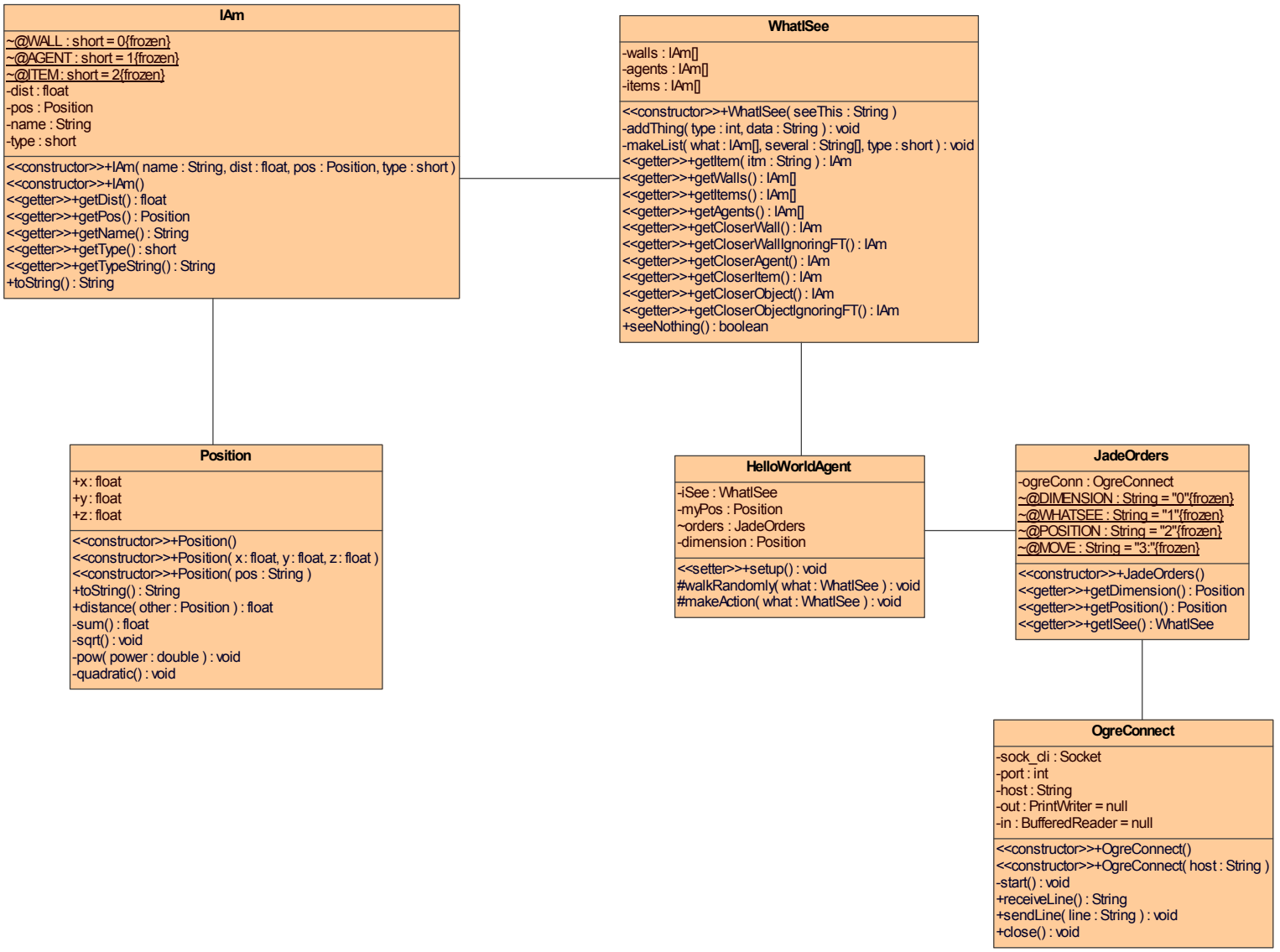
Neste anexo seguem-se e os esquemas UML completos e detalhados das classes do Processamento Gráfico e das classes do Processamento IA.

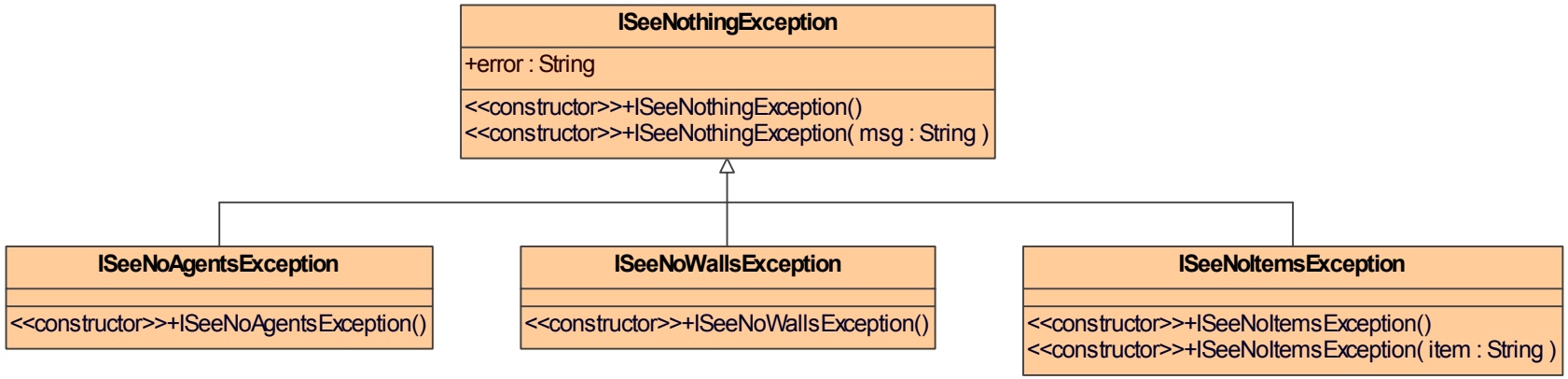
A1. Esquema UML das classes do processamento gráfico




Exceptions
#message : string
+Exceptions(in msg : string)
+~Exceptions()
+getMessage() : string
JadeConnectExceptions
+JadeConnectExceptions(in msg : string)
C ctionLostException
+ConnectionLostException()
InvalidSocketException
+InvalidSocketException ()
+InvalidSocketException (in msg : string)
InvalidServerSocketException
+InvalidServerSocketException ()
InvalidServerSocketException
+InvalidServerSocketException ()
InvalidClientSocketException
+InvalidClientSocketException ()
«struct»IAm
+dist : float
+pos : Point
+name : string
«struct»Point
+x : float
+y : float
+z : float
«datatype»Thing
Auxi
+calculateDist(in a : Vector3, inout b : const Vector3) : float
+getCenter(in no : SceneNode*, in _name : String) : Vector3
+getDirection(in origin : Vector3, in destiny : Vector3) : Vector3
+getStringFromVector3(in pos : Vector3) : string

A2. Esquema UML das classes do processamento de IA




ISeeNothingException
+error : String
<<constructor>>+ISeeNothingException()
<<constructor>>+ISeeNothingException(msg : String)
ISeeNoAgentsException
<<constructor>>+ISeeNoAgentsException()
ISeeNoWallsException
<<constructor>>+ISeeNoWallsException()
ISeeNoItemsException
<<constructor>>+ISeeNoItemsException()
<<constructor>>+ISeeNoItemsException(item : String)

Anexo A3

Neste anexo é efectuada uma descrição pormenorizada das classes mais importantes, descrevendo os seus atributos e métodos mais importantes.

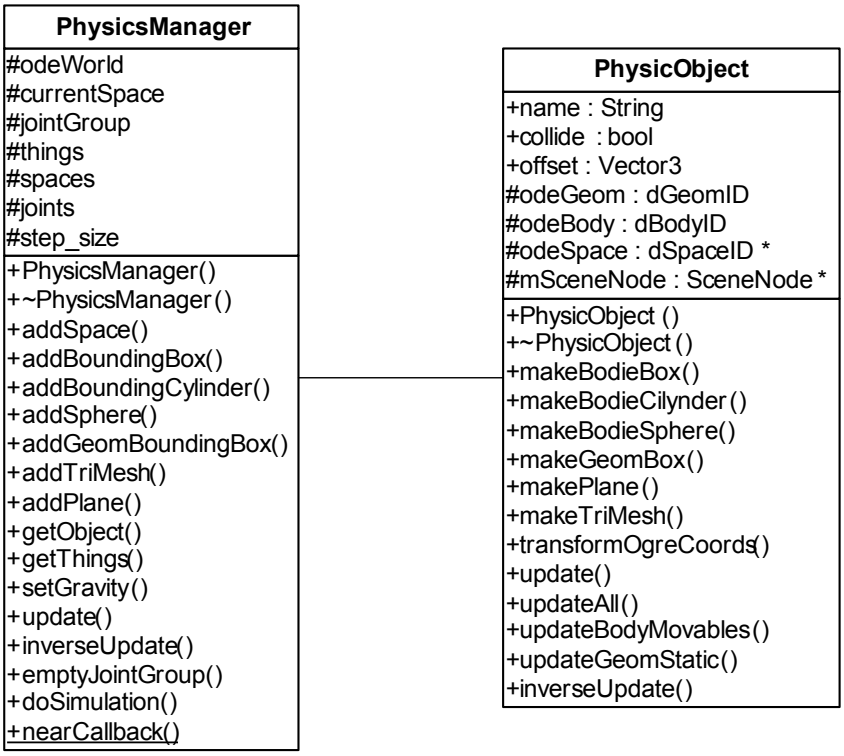


Fig. A3-1 PhysicObject e PhysicsManager

Classe PhysicObject

Sempre que se pretende associar uma simulação de comportamento físico ou de colisões a uma *mesh* (malha poligonal) é necessário criar um objecto desta classe.

Esta classe faz a ponte entre a *mesh* no Ogre e a sua representação física no ODE.

Atributos:

```
String name;
```

Quando se instanciam estas classes, o sistema tem de ser capaz de encontrar e identificar convenientemente a que entidades estão associadas estas instâncias. Para esse efeito, a maioria das classes do PG e do PIA contêm um atributo *name* do tipo *String*, para que o sistema as possa encontrar.

A fim de manter a coerência em todo o sistema, e embora não seja obrigatório, é sensato dar o mesmo valor que se dá ao atributo *name* da classe *Entity* a este objecto. No entanto se o utilizador não der nenhum nome o sistema automaticamente atribui o mesmo valor que se encontra no campo *name* da classe *Entity*. Desta forma o utilizador apenas terá de se preocupar com o valor *name* da classe *Ogre::Entity*.

```
bool collide;
```

Booleano que indica se este objecto está a colidir ou não. A variável tem o valor *true* se estiver a colidir, caso contrário é *false*.

```
Vector3 offset;
```
Centro do objecto. O ODE posiciona os objectos consoante o seu centro. A variável offset serve para se saber qual é o centro do objecto.

```
dGeomID odeGeom;
```
ID da geometria criada no ODE. Sempre que se necessite de ter colisões num determinado objecto, é necessário criar uma geometria no ODE. Após a criação de uma geometria no ODE, é retornado um id que será usado para acessos posteriores a esta geometria.

```
dBodyID odeBody;
```
ID do corpo do objecto. Sempre que se pretende dar uma simulação física a determinado objecto, é necessário criar um corpo ODE com determinadas propriedades. Para se voltar a aceder ao corpo é necessário o seu ID.

```
dSpaceID *odeSpace;
```
Para mais eficientemente se detectar as colisões, os objectos passíveis de colidir devem pertencer a um *odeSpace*. Este atributo indica-nos a que *odeSpace* pertence este objecto, caso ele seja passível de colidir.

```
SceneNode *mSceneNode;
```
Ogre::SceneNode ao qual está associada a *mesh* correspondente.

Métodos:

```
void makeBodieBox();
```
Método responsável pela criação de uma caixa com propriedades físicas e massa associada, a caixa poderá ter propriedades de colisão ou não. Existe a possibilidade de se criar uma caixa "manualmente" ou tendo como base uma malha poligonal do Ogre.

```
void makeBodieCilynder();
```
Semelhante ao *makeBodieBox*(), mas em vez de uma caixa cria um cilindro.

```
void makeBodieSphere();
```
Semelhante ao *makeBodieBox*(), mas em vez de uma caixa cria uma esfera.

```
void makeGeomBox();
```
Método responsável pela criação de uma caixa que contém apenas propriedades de colisão, mas não tem qualquer simulação física. Útil para a criação das colisões de objectos estáticos como paredes, por exemplo.

```
void makePlane();
```
Cria um plano com propriedades de colisão. No ODE os planos apenas têm propriedades de colisão, pois estes são infinitos. Útil para a criação do chão.

```
void makeTriMesh();
```
Cria uma representação da malha poligonal do Ogre recriando-a triângulo a triângulo no ODE. No ODE, quando se monta uma *mesh*, esta apenas pode ter propriedades de colisão e não físicas.

```
void transformOgreCoords();
```
Este método converte as coordenadas da malha poligonal no mundo Ogre por coordenadas do sistema ODE. Calcula também os lados do objecto, tendo como base o *Axis Aligned Bounding Box*, e o centro do objecto.

```
void updateGeomStatic();
```
Este método actualiza as coordenadas ODE tendo como base as coordenadas Ogre caso o objecto seja estático.

```
void updateBodyMovables();
```
Método semelhante ao *updateGeomStatic*() mas apenas actualiza se o objecto for dinâmico.

```
void updateAll();
```
Método que actualiza tudo, da mesma forma que os dois métodos anteriores, independentemente de este ser estático ou dinâmico.

```
void inverseUpdate();
```
Actualiza as coordenadas no sistema ODE tendo como base as coordenadas do objecto no mundo Ogre.

```
void update();
```
Faz um *update* a este objecto. Se estiver em colisão (ou seja se a variável *collide* está a *true*) efectua o *inversUpdate()*. Caso contrário a actualização é feita através do método: *updateBodyMovables()*.

Classe PhysicsManager

PhysicsManager é a classe responsável pela criação e manutenção dos objectos *PhysicObject*. Todos os *PhysicObject*s que se desejam criar devem ser criados através desta classe, caso não seja possível o *PhysicObject* deve ser adicionado à lista *things* desta classe para se manter um historial de todos os *PhysicObject* que estão criados.

Atributos:

```
dWorldID odeWorld;
```
Todos os elementos criados no ODE têm de pertencer a um mundo (*World*) com a finalidade de se fazer a simulação de um mundo. Este atributo é o ID do mundo ODE criado.

```
dSpaceID currentSpace;
```
A fim de se optimizar as simulações de colisão é possível adicionar várias entidades passíveis de colidir no mesmo *ODE::space*. Este atributo guarda o ID do *space* que está a ser usado no momento.

```
dJointGroupID jointGroup;
```
Todas as junções criadas no ODE devem pertencer a um *JointGroup* (grupo de junções). Este atributo guarda o ID do *JointGroup* onde as junções serão criadas.

```
std::list<PhysicObject*>  things;
```
Lista de todos os *PhysicObject* criados. Desta forma é possível saber-se quais e quantos *PhysicObjects* existem no sistema.

```
std::list<dSpaceID> spaces;
```
Lista de todos os espaços ODE (ODE::*Space*) criados. Uma vez que se podem criar vários espaços no ODE com finalidades bem específicas, e também se podem adicionar espaços a outros espaços, desta forma mantêm-se um controlo dos espaços criados.

```
std::list<dJointID> joints;
```
Lista de todas as *joints* criadas.

```
Real step_size;
```
Step da simulação. Sempre que é necessário efectuar uma simulação física o ODE necessita de um passo (*step*).

Métodos:

```
dSpaceID addSpace();
```
Adiciona um ODE *space* à lista *spaces* desta classe e retorna o ID do *space* criado.

```
PhysicObject* addBoundingBox();
```
Cria um *PhysicObject* representando uma caixa com propriedades de colisão e físicas também (como a massa). De seguida adiciona-a à lista de *things*. É possível criar o objecto tendo como base um *SceneNode* do Ogre ou então é criada manualmente. Retorna um apontar para o novo objecto criado.

```
PhysicObject* addBoundingCylinder();
```
Semelhante ao *addBoundingBox()* mas em vez de uma caixa cria um cilindro.

```
PhysicObject* addSphere();
```
Semelhante ao *addBoundingBox*() mas em vez de uma caixa cria uma esfera.

```
PhysicObject* addGeomBoundingBox();
```
Cria um *PhysicObject* com a forma de uma caixa que tem apenas propriedades de colisão e adiciona-o à lista *things*. O *PhysicObject* pode ser construído tendo por base o *SceneNode* do Ogre ou pode ser construído à mão. Retorna o apontador do novo *PhysicObject* criado.

```
PhysicObject* addTriMesh();
```
Semelhante ao *addGeomBoundingBox*() mas em vez de uma caixa cria uma malha poligonal.

```
PhysicObject* addPlane();
```
Cria um *PhysicObject* com a forma de um plano. Retorna um apontador para o novo objecto criado.

```
PhysicObject* getObject();
```
Retorna um apontador para um *PhysicObject* que esteja na lista. A pesquisa pode ser feita tendo como chave de pesquisa (que entra como parâmetro) o ID da geometria do objecto criado, ou o Ogre *SceneNode* que está associado ao objecto, ou pelo nome do *PhysicObject*. Retorna 0 caso não encontre o objecto desejado ou um apontador para o *PhysicObject* encontrado.

```
std::list<PhysicObject*>* getThings();
```
Retorna a lista de todos os *PhysicObject* existentes no sistema.

```
void setGravity();
```
Caso não se pretenda a gravidade definida por omissão (0,0,-0.98) é possível alterá-la através deste método.

```
void update();
```
Faz o *update* a todos os elementos da lista (ver *PhysicObject::update()*).

```
void inverseUpdate();
```
Faz o *inverseUpdate* a todos os elementos da lista *things* (ver *PhysicObject::inverseUpdate()*).

```
void emptyJointGroup();
```
Apaga todas as *joints* criadas.

```
void doSimulation();
```
Efectua uma simulação física e de colisões.

```
static void nearCallback();
```
Esta função é chamada sempre que ocorre uma colisão. Desta forma podemos tratar das colisões consoante os objectos que estão a colidir.

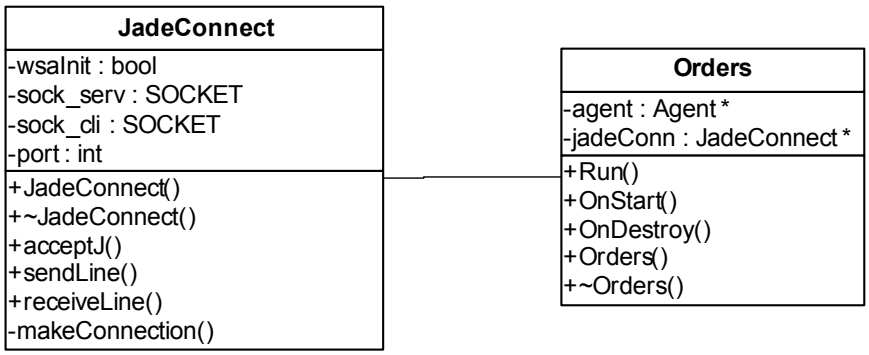


Fig. A3-2JadeConnect e Orders

Classe JadeConnect

Esta classe é equivalente à *OgreConnect* do processamento de IA.

Classe responsável pela ligação, através de *sockets*, ao processamento de IA. Como já foi dito considera-se a hipótese de existir apenas uma conexão do JADE ao Ogre em vez de várias. Esta classe desempenha os papéis de servidor *socket* do sistema.

Atributos:

```
SOCKET sock_serv;
```
Socket do servidor.

```
SOCKET sock_cli;
```
Socket onde o cliente se liga.

```
int port;
```
Porto onde o *socket* do servidor (*sock_serv*) fica à escuta de ligações. Por *default* o porto é 30001. Mas pode ser alterado quando se cria o objecto.

Métodos:

```
void acceptJ();
```
Aceita uma ligação e associa-lhe o *sock_cli*.

```
void sendLine ();
```
Envia uma *string* pela *sock_cli*.

```
std::string receiveLine();
```
Recebe uma *string* pela *sock_cli* e retorna-a.

```
void makeConnection();
```
Estabelece a ligação. Inicializa a *sock_serv*, faz-lhe o *binding* e fica à escuta de ligações.

Classe Orders

Classe responsável pela criação e destruição do *JadeConnect*. Esta classe estende a *CCoreConcurrentObject* que cria uma nova *thread* que irá correr em ciclo no método *run()*. *CcoreConcurrentObject* é uma classe que pretende encapsular a criação de novas *threads*, sendo a mesma transparente para o utilizador que crie a classe *Orders*.

Orders espera por mensagens vindas do cliente e processa-as.

Atributos:

```
Agent* agent;
```
Apontador para o agente que se irá movimentar e receber certas ordens. Consultar a secção 5. Em trabalho futuro pretende-se adicionar uma classe do tipo *AgentsManager* que à semelhança da *PhysicsManager* trata da criação e mantém controlo sobre todos os agentes da bancada Ogre.

```
JadeConnect *jadeConn;
```
Apontador para o objecto *JadeConnect*. Como já foi referido esta classe é responsável pela criação e destruição de *JadeConnect*.

Métodos:

```
void OnStart();
```
Método que é chamado assim que a *thread* vai correr neste objecto. Este método executa as instruções necessárias ao início da execução da *thread*. Neste caso fica à espera de receber uma conexão em *jadeConn*.

```
void Run();
```
Método que corre em ciclo infinito na *thread* criada. É neste método que se escutam por novos pedidos do cliente que depois são executados.

```
void OnDestroy();
```
Este método é chamado quando se destrói um objecto desta classe. Executa as instruções necessárias antes de matar a *thread*.

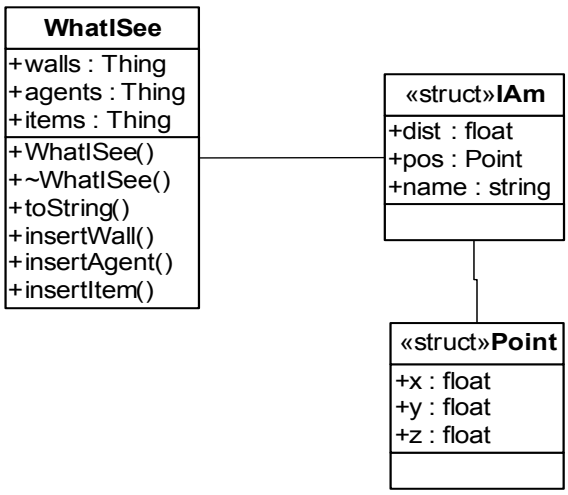


Fig. A3-3 WhatISee, IAm e Point

Classe Point

Apesar de em C++ (processamento gráfico) esta classe ser uma estrutura é em tudo semelhante à classe *Position* do processamento de IA. Classe que guarda três valores reais, um para cada coordenada x, y e z. Esta classe pode representar um ponto ou posição no espaço bem como a dimensão de uma caixa.

Atributos:

`float X;`
Coordenada x do ponto.

`float y`
Coordenada y do ponto.

`float z`
Coordenada z do ponto.

Classe IAm

Estrutura simples que guarda certos dados importantes de um determinado objecto. É equivalente à classe *IAm* pertencente ao processamento de IA.

Atributos:

`float dist`
Distância deste objecto a um determinado agente

`Point pos;`
Posição do objecto no espaço

`string name;`
Nome do objecto.

Classe WhatISee

Classe que guarda a informação sobre o que determinado agente vê.

Atributos:

```
Thing walls;
```

Vector que contém elementos do tipo *IAm*. Todas as paredes que o agente está a ver.

```
Thing agents;
```

Todos os agentes que o agente está a ver.

```
Thing items;
```

Todos os itens que o agente está a ver.

Métodos:

```
void insertWall();
```

Insere uma parede no vector *walls*.

```
void insertAgent();
```

Insere um agente no vector *agents*.

```
void insertItem();
```

Insere um item no vector de *items*.

```
std::string toString();
```

Devolve uma *string* que representa este mesmo objecto. É utilizado quando se pretende enviar a informação pela *socket*.

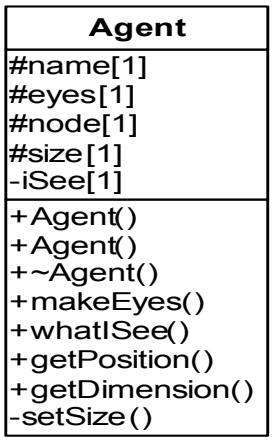


Fig. A3-4 Agent

Classe Agent

Classe que representa um agente inteligente. Um agente tem certas características como os seus sensores (olhos), tamanho, etc. que estão representadas nesta classe.

Atributos:

```
String name;
```

Nome do agente. Tem que ser igual ao nome correspondente na camada de PIA.

A fim de manter a coerência em todo o sistema, e embora não seja obrigatório, é sensato dar o mesmo valor que se dá ao atributo *name* da classe *Entity* a este objecto. Se o utilizador não der nenhum nome o sistema automaticamente atribui o mesmo valor que se encontra no campo *name* da classe *Entity*. Desta forma o utilizador apenas terá de se preocupar com o valor *name* da classe *Ogre::Entity*.

```
Camera *eyes;
```

Os olhos do agente são simulados através de uma câmara montada na sua cabeça.

```
SceneNode *node;
```

SceneNode do Ogre que corresponde ao objecto móvel que é este agente.

```
Vector3 size;
```

Tamanho do agente.

```
WhatISee *iSee;
```

O que o agente vê.

Métodos:

```
Camera* makeEyes();
```

Método que cria os "olhos" do agente.

```
WhatISee* whatISee();
```

Método que retorna um apontador para um objecto do tipo *WhatISee* que contêm informação acerca do que o agente está a ver no momento.

```
std::string getPosition();
```

Retorna uma *string* representando a posição do agente.

```
std::string getDimension();
```

Retorna uma *string* representando a posição do agente.

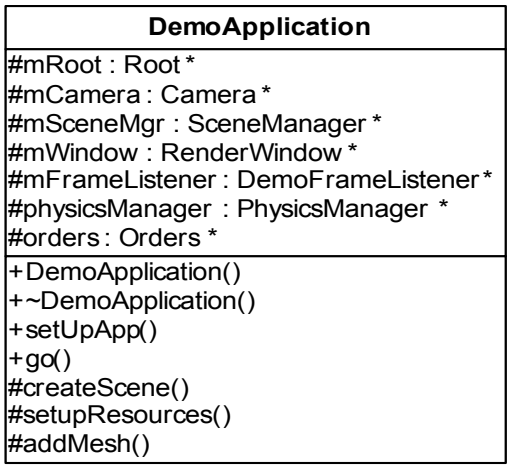


Fig. A3-5 DemoApplication

Classe DemoApplication

Esta é a classe inicial, onde tudo começa. Os ficheiros extraídos do Blender são carregados para o motor gráfico através de objectos desta classe. As câmaras, as opções de *rendering*, as luzes, etc. são todos inicializados através de objectos desta classe.

Atributos:

```
Root *mRoot;
```

A *Ogre::Root* classe representa o ponto de partida para as aplicações que usam o Ogre *framework*. A partir daqui ganha-se o acesso aos pontos fundamentais do sistema, nomeadamente ao sistema de *rendering* disponível, às configurações gravadas, torna-se possível carregar e aceder a outras classes do sistema. Uma instância de *Root* deve ser criada antes de qualquer outra operação Ogre.

```
SceneManager *mSceneMgr;
```

Ogre::SceneManager é a classe responsável pela organização da cujo *rendering* será feito mais tarde.

```
Camera *mCamera;
```

Ogre faz o *rendering* da cena a partir de uma câmara. Esta é a câmara principal e que afecta toda a janela do Ogre.

```
RenderWindow *mWindow;
```

Esta classe trata da janela na qual os conteúdos de uma cena serão desenhados.

```
DemoFrameListener *mFrameListener;
```

Classe que estende a classe *Ogre::FrameListener*. *Ogre::FrameListener* é uma interface desenhada para ser chamada quando um evento particular acontece (inicio da frame, fim da frame). É aqui que estão definidos os controlos de *input* do teclado, bem como todas as parametrizações necessárias quando se começa uma frame ou quando acaba. Cada *Ogre::FrameListener* deve ser adicionada ao objecto *Root* criado.

```
PhysicsManager *physicsManager;
```
Classe responsável pela criação e manutenção dos objectos físicos. Quando se pretende que determinada *mesh* (malha poligonal) tenha propriedades de colisão e físicas é necessário criar um objecto do tipo *PhysicObject*.

```
Orders* orders;
```
Classe responsável pela *socket*. Esta classe fica à escuta num porto e executa os pedidos que lhe são feitos.

Métodos:

```
bool setUpApp();
```
Método inicial. Método responsável pela configuração de todo o sistema e inicialização de todas as classes necessárias à aplicação de *rendering*. Se algo correr mal, retorna *false*, caso contrário retorna *true*.

```
void go();
```
Caso o método *setUpApp()* tenha sucesso, este é o método responsável pelo começo do *rendering*.

```
void DemoApplication::createScene();
```
Método responsável pela criação e montagem da cena. Neste método são criadas as instancias das malhas poligonais criadas no Blender, bem como a sua posição no espaço. É também aqui que se criam e definem as luzes da cena.

```
void setupResources();
```
Método responsável por carregar os *resources* que o Ogre vai usar. Entenda-se por *resources*, todos os ficheiros exportados do Blender, todas as texturas, materiais, *overlays*, entre outros.

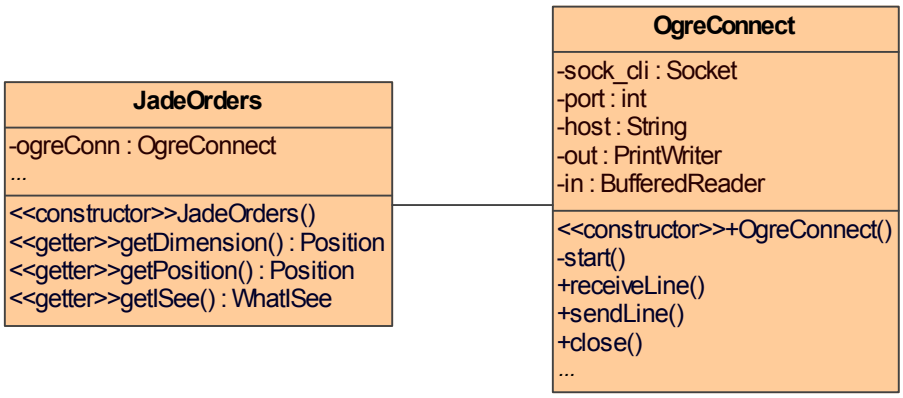


Fig. A3-6 JadeOrders e OgreConnect

Classe OgreConnect

Esta classe é semelhante à classe *JadeConnect* no sentido em que é responsável pelo estabelecimento da ligação da *socket* ao servidor. No entanto esta é uma *socket* de cliente e não de servidor.

Atributos:

`Socket sock_cli`
A *socket* cliente que vai estabelecer a ligação TCP ao servidor.

`int port;`
Porto sobre o qual o servidor está à escuta.

`String host;`
Endereço do servidor.

`PrintWriter out;`
Buffer de escrita. Este *buffer* usa a *socket* para enviar *strings* de caracteres.

`BufferedReader in;`
Buffer de leitura. Este *buffer* usa a *socket* para receber *strings* de caracteres do servidor.

Métodos:

```
void start();
```
Método responsável pela inicialização da *socket* e dos *buffers* de escrita e leitura.

```
String receiveLine();
```
Método que recebe uma *string* de caracteres do servidor.

```
void sendLine();
```
Método que envia uma *string* de caracteres para o servidor.

```
void close();
```
Método que fecha a *socket* e os *buffers* de leitura e escrita.

Classe JadeOrders

Classe responsável por enviar pedidos ao servidor e receber as respostas deste.

Atributos:

```
OgreConnect ogreConn;
```
Classe *OgreConnect* responsável pela ligação ao servidor.

Métodos:

```
Position getDimension();
```
Método para obter a dimensão do agente.

```
Position getPosition();
```
Método para obter a posição do agente.

```
WhatISee getISee();
```
Método para obter o que o agente está a ver.

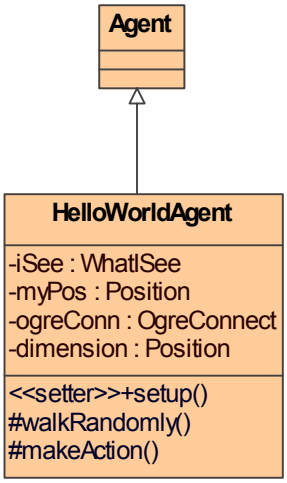


Fig. A3-7 HelloWorldAgent e Agent

Classe HelloWorldAgent

Os agentes que se constroem usando a bancada JADE têm que estender a classe *Agent* pertencente a essa bancada. Desta forma os agentes herdam as primitivas e funcionalidades básicas dos agentes da bancada JADE. Esta classe serve essencialmente para se fazer experiências e testes à arquitectura e representa um protótipo do que será um agente futuramente.

Atributos:

```
WhatISee iSee;
```

O que o agente está a ver. Esta classe já foi descrita.

```
Position myPos;
```

A posição actual do agente.

```
ogreConn OgreConnect;
```

Classe que trata da ligação ao servidor e do encaminhamento dos pedidos.

```
Position dimension
```

Dimensão do agente.

Métodos:

```
void WalkRandomly();
```

Este método faz com que o agente ande aleatoriamente pelo cenário evitando colidir com os objectos.

```
void makeAction();
```

Método usado para obrigar o agente a fazer uma acção. Esta acção pode ser desde andar em frente virar a cabeça, etc.

Anexo B

Neste anexo enumeram-se as capacidades das ferramentas Blender, Ogre e ODE.

B1. Blender

Indicam-se seguidamente as principais funcionalidades do Blender.

Modelação

- Permite a manipulação de: NURBS, curvas de Bezier e *B-splines, metaballs, vector fonts* (*TrueType, PostScript, OpenType*)
- *'Smooth proxy' style catmull-clark subdivision surfaces*
- Funções Booleanas em malhas
- Funções de edição tais como: *extrude, bevel, cut, spin, screw, warp, subdivide, noise, smooth Soft selection editing tools for organic modeling*
- Criação de ferramentas de modelação através de *scripts* Python

Animação

- *Armature (skeleton) deformation with forward/inverse kinematics, auto skinning and interactive 3D paint for vertex weighting*
- *Non-linear animation mixer with automated walkcycles along paths*
- *Constraint system*
- *Morphing* com *vertex key framing* controlado por *sliders*
- Editor de *character animation pose*
- *Animatable lattice deformation*
- Sistema 'Ipo' integrando *motion curve* e *traditional key-frame editing*
- Facilidades de sincronização de som
- Efeitos de animação produzidos através de *scripts* Python

Criação de Jogos 3D em tempo Real

- Editor gráfico para definir o comportamento sem necessidade de programação.
- Detecção de colisões
- *Scripts* Python para control de IA, lógica do jogo completamente definida
- Suporta todos os modos de iluminação do OpenGL™, incluindo transparências. Texturas
- *Playback* de jogos e conteúdo 3D sem compilação ou preprocessamento
- Audio, usando a ferramenta the fmod
- *Multi-layering* de cenas

Rendering

- *Raytracer* incorporado
- Suporta o motor de rendering Yafray
- *Oversampling, motion blur, post-production effects, fields, non-square pixels*
- *Environment maps, halos, lens flares, fog*
- Concretiza vários algoritmos de shading: Lambert, Phong, Oren-nayar, Blinn, Toon
- *Edge rendering for Toon shading*
- *Procedural Textures*
- *Ambient Occlusion*

- *Radiosity solver*
- *Scripts* para exportação *renderers* externos como Renderman (RIB), Povray, Virtualight
- *UV texture editor*

Interface

- Disposição de janelas configurável pelo utilizador
- Sistema de dados orientado por objectos
- Janelas para: animação por curvas ou pontos, diagramas esquemáticos de cenas, edição de sequências de vídeo não lineares, editor para animação da acção dos personagens, misturador de animação não linear, edição de image/UV, selecção de imagens ou ficheiros e gestor de ficheiros
- Editor de texto para anotação e edição de *scripts* Python
- Interface coerente em diferentes plataformas

Ficheiros

- Guarda todos os dados da cena num único ficheiro .blend
- O formato .blend suporta compressão, assinaturas digitais, encriptação, compatibilidade *forwards/backwards* e pode ser usado como uma biblioteca para para ligar com outros ficheiros .blend.
- Suporta os formatos TGA, JPG, PNG, Iris, SGI Movie, IFF, AVI, Quicktime, GIF, TIFF, PSD, MOV (Windows and Mac OS X)
- Importa e exporta ficheiros DXF, Inventor e VRML com *scripts* Python também disponíveis para outros formatos 3D.
- Criação de executáveis *stand-alone* com conteúdo 3D interactivo.

B2. Ogre

Indicam-se seguidamente as principais funcionalidades do Blender.

Características de Produtividade

- Uso simples de uma interface OO (Orientada a Objectos) desenhada para minimizar o esforço de *rendering* de cenas 3D, e idependente da implementação 3D i.e. Direct3D ou OpenGL.
- *FrameWork* que pode ser estendida por fomr a se fazer aplicações de forma rápida e simples.
- Requerimentos comuns como gerenciador de rendering, cortes hierarquicos, manusemento de transparencias são feitos automáticamente, poupando valioso de desenvolvimento tempo.
- Todas as classes do motor estão desenhadas de forma clara e completamente documentadas.

Plataforma & API de Suporte 3D

- Suporte Direct3D e OpenGL
- Windows (todas as versões), Linux e Mac OSX
- Compila em Visual C++ 6 (com STLport), Visual C++.Net 2002 (com STLport), Visual C++.Net 2003 em Windows
- Compila em gcc 3+ em Linux / Mac OSX (usando XCode)

Suporte Material / Shader

- Linguagem especifica e poderosa para editar/fazer novos materiais fora do código.
- Suport a *vertex* e *fragment programs* (*shaders*), ambas as linguagens de baixo nivel escritas em assembler, e linguagens de mais alto nível escritas em Cg ou DirectX9 HLSL.
- *Provides automatic support for many commonly bound constant parameters like worldview matrices, light state information, object space eye position etc*

- Suporta um leque completo de operações como *multitexture* e *multipass blending*, *texture coordinate generation and modification*, *independent colour and alpha operations for non-programmable hardware or for lower cost materials*
- *Multiple pass effects, with pass iteration if required for the closest 'n' lights*
- Suporte a multiplas téqnicas de material, o que significa que se podem desenhar para uma variadade grande de técnicas que o OGRE automáticamente escolhe qual a melhor configuração para a placa gráfica em uso.
- Suporte a *Material LOD*;
- Carrega texturas dos seguintes formatos: PNG, JPEG, TGA, BMP ou DDS, incluindo formatos menos usuais como texturas ID, *volumetric textures*, *cubemaps* e texturas comprimidas (DXT/S3TC)
- Texturas podem ser provenientes e actualizadas em tempo real por *plugins*.
- *Easy to use projective texturing support*

Malhas Poligonais

- Formato fléxivel de malhas poligonais, separação dos conceitos de *vertex buffers*, *index buffers*, *vertex declarations* e *buffer mappings*
- Exportação a partir de muitas ferramentas de modelação incluindo: Milkshape3D, 3D Studio Max, Maya, Blender e Wings3D
- Animação por esqueleto, incluindo o *blending* de múltiplas animações, distribuição de pesos variável por osso e skinning acelarado por *hardware*.
- *Biquadric Bezier patches for curved surfaces*
- Malhas poligonais progressivas (LOD)

Caracteristicas da Cena

- Gerenciador da cena altamente costumisavel e flexivel, usando vários tipos de cena. Uso de classes predefenidas para a organização da cena ou estendem-se as classes ganhando controlo absoluto sobre a organização da cena.
- Vários *plugins* de exemplo para demonstrar as várias maneiras de manipular a cena para um tipo especifico de *layout* (e.x. BSP, Octree)
- Grafo da cena hierárquica; os nós permitem que os objectos a eles associados sigam todos os seus movimentos.
- Múltiplas técnicas de rendering de sombras, cada uma altamente configurável e tirando todo o partido de aceleração por *HardWare*.
- *Scene querying features*

Efeitos Especiais

- Sistemas de Particulas, incluindo emissores e *affectors* estensiveis (costumizaveis atraves de *plugins*). Os sistemas podem ser definidos em texto (scripts) para uma afinação mais fácil. Uso automático de *pooling* de partículas para uma performance máxima.
- Suporte a *skyboxes*, *skyplanes* e *skydomes*, muito fáceis de usar
- *Billboarding for sprite graphics*
- Controlo automático de objectos transparentes (oredem de *rendering* e ajustes do *buffer* de profundidade automáticos)

Características Várias

- Infraestrutura comum de recursos para gerenciamento de memória e carregamento dos arquivos (ZIP, PK3)
- Arquitectura de p*lugins* flexivel permitindo a estenção do motor sem a necessidade de recompilar todo o motor.

- Controladores permitindo uma organização fácil dos valores derivados entre objectos e.x. mudança da cor de uma nave baseada no escudo esquerdo.
- *Debugging memory manager for identifying memory leaks*
- A ReferenceAppLayer fornece um exemplo sobre como combinar o Ogre com outras bibliotecas, por exemplo ODE para colisões e física.
- *Converter to convert efficient runtime binary formats to/from XML for interchange or editing*

B3. ODE

Indicam-se seguidamente as principais funcionalidades do ODE.

Características Gerais

Basic Physics, Collision Detection, Rigid Body, Vehicle Physics:

- Corpos Rigidos (*Rigid bodies*) com massa distribuida arbitrariamente.
- Tipos de junções (*joint*): *ball-and-socket*, *hinge*, *slider (prismatic)*, *hinge-2*, *fixed*, *angular motor*, *universal*.
- Primitivas de Colisão: *sphere*, *box*, *capped cylinder*, *plane*, *ray*, e *triangular mesh*.
- *Collision spaces: Quad tree, hash space, and simple.*
- Método de simulação: as equações de movimento são derivadas a partir de um modelo baseado no *multiplier* velocity de Lagrange.
- É usado um integrador de primeira ordem. É rápido, mas ainda não é muito exato para engenharia quantitativa. Integradores de ordens superiores estão a ser desenvolvidos.
- Escolha do tempo de *step*. Este pode ser *standard* (matriz grande) ou um novo e interactivo método *QuickStep* pode ser usado.
- Modelo de contacto e fricção: é baseado no *Dantzig LCP* descrito por Baraff, no entanto ODE implementa uma aproximação mais rápida de fricção usando o modelo Coloumb.
- Optimizações específicas consoante a plataforma.